Visual Assembly and Analysis of Cryptographic Ciphers

Carlos Jerome Tafoya
University of Colorado at Boulder, cjtafoya512@hotmail.com
VISUAL ASSEMBLY AND ANALYSIS
OF CRYPTOGRAPHIC CIPHERS

by

CARLOS JEROME TAFOYA

B.S., University of New Mexico 2009

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirement for the degree of
Master of Science
Department of Computer Science
2011
This thesis entitled:
Visual Assembly and Analysis of Cryptographic Ciphers
written by Carlos Jerome Tafoya
has been approved for the Department of Computer Science

__________________________________________________________
John Black

__________________________________________________________
Andrzej Ehrenfeucht

Date______________

The final copy of this thesis has been examined by the signatories, and we
Find that both the content and the form meet acceptable presentation standards
Of scholarly work in the above mentioned discipline.
Cryptography can be viewed as the science of information hiding. One key primitive of cryptography is the cipher. In order to assemble and analyze modern ciphers, cryptographers must be well versed in mathematics and computer science. This intersection of knowledge is uncommon and thus cryptography remains a field with a small population. Through the use of visual programming we will attempt to make this field more main stream. CryptKeeper is a visual programming tool designed to make the assembly and analysis of ciphers easier. It is designed to be a supplementary tool to available prospecting cryptographers. CryptKeeper eliminates many of the complexities inherent to programming modern ciphers, and handles much of the mathematics required. It allows users to visually watch the execution of ciphers and analyze how they process data. CryptKeeper allows users to focus more on the details of the cipher rather than the details of its implementation.
CONTENTS

1 INTRODUCTION ......................................................................................................................... 1
  1.1 VISUAL PROGRAMMING ................................................................. 1
  1.2 THESIS SUMMARY .............................................................................. 2

2 PRELIMINARIES ......................................................................................................................... 4
  2.1 BIT STRINGS ......................................................................................... 4
  2.2 CIPHERS ................................................................................................. 6
  2.3 ABSTRACT ALGEBRA .......................................................................... 7
    2.3.1 Groups ............................................................................................ 7
    2.3.2 Rings ................................................................................................ 8
    2.3.3 Fields ................................................................................................ 9

3 CRYPTKEEPER ....................................................................................................................... 11
  3.1 USER INTERFACE ................................................................................ 11
    3.1.1 CryptKeeper Window ................................................................. 12
    3.1.2 Palette ............................................................................................ 13
    3.1.3 Property Viewer ............................................................................ 14
    3.1.4 Console ........................................................................................... 15
  3.2 CRYPTKEEPER PROGRAMS ............................................................. 15
    3.2.1 Code Nodes .................................................................................. 16
    3.2.2 Signals ........................................................................................... 24
    3.2.3 Modules ........................................................................................ 24
    3.2.4 Program Execution ....................................................................... 25
  3.3 EXAMPLE ..................................................................................................... 25

4 DES ASSEMBLY AND ANALYSIS ....................................................................................... 27
  4.1 THE DES FUNCTION .......................................................................... 27
  4.2 THE DES ROUND ............................................................................... 30
  4.3 DES ALGORITHM ................................................................................ 31

5 AES ASSEMBLY AND ANALYSIS ..................................................................................... 33
  5.1 THE STATE ......................................................................................... 33
  5.2 SUBBYTES ............................................................................................. 33
  5.3 SHIFT ROW ........................................................................................... 34
  5.4 MIXCOLUMN .......................................................................................... 35
  5.5 ADDROUNDKEY ................................................................................... 35
  5.6 AES ROUND ........................................................................................ 35
  5.7 AES FINAL ROUND ........................................................................... 37
  5.8 AES ALGORITHM ................................................................................. 37

6 SALSA20 ASSEMBLY AND ANALYSIS ............................................................................. 39
  6.1 QUARTER ROUND ............................................................................... 39
  6.2 ROW ROUND ....................................................................................... 41
  6.3 COLUMN ROUND ............................................................................... 43
  6.4 DOUBLE ROUND ................................................................................... 44
  6.5 LITTLE ENDIAN .................................................................................... 45
  6.6 MATRIX LITTLE ENDIAN ..................................................................... 46
  6.7 SALSA20 HASH .................................................................................... 47
  6.8 SALSA20 EXPANSION ........................................................................ 49
  6.9 SALSA20 ENCRYPTION .................................................................... 51

7 CONCLUSION ......................................................................................................................... 53
LIST OF TABLES

TABLE 4-1 E PERMUTATION TABLE ................................................................. 28
TABLE 4-2 LOOKUP TABLE FOR S1 ............................................................. 29
TABLE 4-3 THE P PERMUTATION TABLE .................................................... 29
TABLE 4-4 IP PERMUTATION TABLE ............................................................ 32
TABLE 4-5 FP PERMUTATION TABLE .......................................................... 32
TABLE 5-1 SUBBYTES LOOKUP TABLE ..................................................... 34
TABLE 5-2 SHIFTRow PERMUTATION ......................................................... 35
TABLE 6-1 ROWROUNDS PERMUTATION TABLE ....................................... 43
TABLE 6-2 ROWROUND INVERSE PERMUTATION ...................................... 43
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>THE CRYPTKEEPER WINDOW</td>
<td>12</td>
</tr>
<tr>
<td>3-2</td>
<td>THE PALETTE</td>
<td>14</td>
</tr>
<tr>
<td>3-3</td>
<td>THE PROPERTY VIEWER</td>
<td>14</td>
</tr>
<tr>
<td>3-4</td>
<td>HELLO VALUE</td>
<td>26</td>
</tr>
<tr>
<td>4-1</td>
<td>THE F FUNCTION</td>
<td>28</td>
</tr>
<tr>
<td>4-2</td>
<td>THE DES ROUND FUNCTION</td>
<td>30</td>
</tr>
<tr>
<td>4-3</td>
<td>THE FULL DES ALGORITHM</td>
<td>31</td>
</tr>
<tr>
<td>5-1</td>
<td>AES ROUND</td>
<td>36</td>
</tr>
<tr>
<td>5-2</td>
<td>AES FINAL ROUND</td>
<td>37</td>
</tr>
<tr>
<td>5-3</td>
<td>AES ALGORITHM</td>
<td>38</td>
</tr>
<tr>
<td>6-1</td>
<td>QUATERROUND</td>
<td>40</td>
</tr>
<tr>
<td>6-2</td>
<td>ROWROUND</td>
<td>42</td>
</tr>
<tr>
<td>6-3</td>
<td>COLUMN ROUND</td>
<td>44</td>
</tr>
<tr>
<td>6-4</td>
<td>DOUBLE ROUND</td>
<td>45</td>
</tr>
<tr>
<td>6-5</td>
<td>LITTLE ENDIAN</td>
<td>46</td>
</tr>
<tr>
<td>6-6</td>
<td>MATRIX LITTLE ENDIAN</td>
<td>47</td>
</tr>
<tr>
<td>6-7</td>
<td>SALSA20 HASH</td>
<td>48</td>
</tr>
<tr>
<td>6-8</td>
<td>THE SALSA EXPANSION FUNCTION</td>
<td>50</td>
</tr>
<tr>
<td>6-9</td>
<td>SALSA20 ENCRYPTION FUNCTION</td>
<td>51</td>
</tr>
</tbody>
</table>
1 Introduction

Cryptography can be viewed as the science of information hiding. Cryptography has applications that intersect many aspects of modern day life. From electronic commerce to ATM cards, cryptography is prominent in our daily activities. Despite its prominence in the world around us, cryptography is viewed by many as a daunting field of study. One key primitive of cryptography is the cipher. In order to assemble and analyze modern ciphers, cryptographers must be well versed in both mathematics and computer science. This intersection of knowledge is uncommon and thus cryptography remains a field with a small population. Through the use of graphical programming we will attempt to make this field more mainstream.

This paper will introduce the CryptKeeper visual programming environment. This tool is intended to make the assembly and analysis of ciphers easier. It is designed to be a supplementary tool available to prospecting cryptographers. CryptKeeper eliminates many of the complexities inherent to programming modern ciphers, and handles much of the mathematics required. CryptKeeper allows users to visually watch the execution of ciphers and analyze how they process data. CryptKeeper allows users to focus more on the details of the cipher rather than the details of its implementation.

1.1 Visual Programming

Visual programming languages and their associated programming environments were born from a desire to make programming easier to learn and appeal to a wider audience [1]. Text based programming languages like C or Java, while powerful, can be difficult to master. Visual Programming is a type of programming that lets users generate
a program by manipulating the various components of a program graphically; rather than using the standard approach of specifying a program textually.

A common practice of visual programming is to represent the program as a directed graph. The nodes of the graph represent the behavior of the program while the edges represent the flow of data. CryptKeeper follows this paradigm. It uses nodes (also called code nodes) which operate on data, and signals which direct the flow of data from one node to another. CryptKeeper however is not intended to be a general purpose programming language. It is domain specific to the assembly and analysis of cryptographic ciphers.

1.2 Thesis Summary

This paper will explore the power and potential of the CryptKeeper system. Throughout the paper we will assemble and analyze several cryptographic ciphers. We will implement these ciphers in CryptKeeper and use the implementation as a basis for our analysis.

Chapter 2 will cover some basic preliminaries needed to better understand cryptographic ciphers. It will informally define what a cipher is and cover some needed mathematics. All explanations in this chapter are intended to be simple introductions to the material.

Chapter 3 will describe some design choices made in developing CryptKeeper. It will also introduce the basic building blocks of a CryptKeeper program. The User interface will be introduced as well, along with simple descriptions of the various code nodes available. This is not meant to be a complete documentation of the CryptKeeper
system, but rather a crash course so that we understand the system well enough to use it in our assembly and analysis of various ciphers.

In chapters 4 thru 6 we will demonstrate CryptKeepers usefulness in assembling and analyzing various ciphers. We will explore the Data Encryption Standard (DES), due to its historical significance then move on to the current encryption standard, the Advanced Encryption Standard (AES). We will conclude our assembly and analysis with the Salsa20 encryption algorithm.
2 Preliminaries

Before delving into the main topic of the thesis, visual assembly and analysis of cryptographic ciphers, we will introduce some mathematics, notation and conventions that will be used throughout the paper. These will all be introduced in a simple manner. The following is not a comprehensive description of what is needed to do cryptography, but rather a selection that will be useful for the confines of this paper. It will be assumed that readers have a basic understanding of set theory and computer science.

2.1 Bit Strings

In computer science, a string is defined as a sequence of characters. These characters are generally taking from a set of elements called the alphabet. The set \{0,1\} defines the alphabet used by bit strings. This alphabet is the set of binary digits. In general a bit string will be an element of the set \{0,1\}^+ (bit strings contain one or more characters or bits). The set \{0,1\}^n is the set of bit string that contains exactly n-bits where \(n > 0\). If \(S \in \{0,1\}^+\) (S is a bit-string) then \(S_i\) will be the \(i^{th}\) bit of the bit string. If \(S\) is \(n\)-bits long, then we will index the bits of \(S\) from 0 to \((n-1)\) from left to right. If \(i\) is greater than or equal to \(n\), then we say that \(S_i\) is undefined. For example let \(S = \{0,1,1,0\}\) be a bit string of length 4, then: \(S_0 = 0, S_1 = 1, S_2 = 1, and S_3 = 0\). We will sometimes call \(S_0\) bit 0 or the first bit, and \(S_n\) bit \(n\) or the last bit.

We have shown the set notation for defining a bit string, but will sometimes use an alternate form. The binary string “0b1110” is equivalent to the bit string \{1,1,1,0\}. 
When the context is clear we will leave off the “0b” header and write the bit string as “1110”. As a short had for writing bit strings we may also use decimal or hexadecimal version. The bit string “1110” is equivalent to decimal string “14” or hexadecimal string “0xE”. When the meaning is clear we will exclude the hexadecimal header “0x”.

We will allow various arithmetic operations on bit strings. When an arithmetic operation is being applied to bit strings they will be interpreted as unsigned integer values. In these cases the bit string will be interpreted as an integer in base 2. Arithmetic operations can never cause a bit string to overflow or underflow. If the bit string has been defined to exist in a finite field, the arithmetic operations will obey the rules of the field. Section 2.3.3 will discuss fields.

We allow logical operations to be applied to bit strings. In the case of binary operators if the bit strings being applied have different lengths, the shorter bit string will have zeros added to the front of its bits until it matches the length of the larger bit string. This will be termed zero extending. After the bit strings are of equal length, the operation will be performed on a bit by bit basis. For example, Let \( X = \{0,1,0,1\} \) and \( Y = \{1,1,0,1,0\} \), then \( X \oplus Y \) (X xor Y) will cause X to be zero extended and then the operation performed. So \( X \oplus Y = \{1,1,1,1,1\} \)

We will also allow relation comparisons of bit strings. In these cases the bit strings will be interpreted as unsigned integers. Results of these comparisons will be either the bit string \{0\} or the bit string \{1\}. The bit string \{0\} represents false, while \{1\} represents true. For example let \( X = \{1,0\} \) or 2 in decimal and \( Y = \{1,1\} \) or 3 in decimal then \( X < Y = \{1\} \) or true
2.2 Ciphers

In cryptography a cipher is an algorithm for performing encryption and/or decryption [2]. When using a cipher for encryption the original information is known as plain text and the enciphered information is known as cipher text. When a cipher is used for decryption it is given cipher text and produces plain text. Plain text is data that is in an understandable form while cipher text is that same information but in a format that is not readily understandable. Only by using the cipher to decrypt to cipher text will the information become understandable. Ciphers generally depend on a supporting piece of information known as the key. Given the same plain text, but different keys a cipher should produce different versions of cipher text. In order to decrypt the cipher text a user of the cipher must have the same key (or a related key) that was used to encrypt it. Without this key it should be nearly impossible to decrypt the cipher text.

Modern ciphers can be categorized into symmetric key and asymmetric key algorithms. In symmetric key algorithms the sender and receiver of an encrypted message share the same key. In asymmetric key algorithms separate keys will be used for encryption and decryption. In this scheme one key is public knowledge. For this paper we will only concern ourselves with symmetric key algorithms. Symmetric key algorithm ciphers can be further broken up into two main categories, block ciphers and stream ciphers.

A block cipher can be defined as: \( F : \{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^n \), that is a function that takes a n-bit plain text and a k-bit key, and outputs a n-bit cipher text [3]. In this paper we will explore the DES and AES block ciphers.
A stream cipher can be viewed as a pseudorandom bit stream generator [4]. When using a stream cipher the plain text bits are generally xor-ed with the bits generated by the stream cipher. In this paper we will explore the Salsa20 stream cipher.

2.3 Abstract Algebra

Groups Rings and Fields are related mathematical constructs that will be needed during our discussion of the AES block cipher [5]. What follows is only a brief, and incomplete, introduction to these concepts.

2.3.1 Groups

A group can be viewed as a set and an associated operation that satisfy certain rules. A group can be either an infinite set of elements of a finite set. A nonempty set $G$ is said to be a group if in $G$ there exists an operation, denoted “$*$”, such that:

- $a, b \in G \rightarrow a \ast b \in G$ (we say that $G$ is closed under $*$)
- $a, b, c \in G \rightarrow a \ast (b \ast c) = (a \ast b) \ast c$ (the associative law holds in $G$)
- $\exists e \in G : a \ast e = e \ast a = a, \forall a \in G$ (there exists an identity element)
- $\forall a \in G, \exists b \in G : a \ast b = b \ast a = e$ (all elements in $G$ have an inverse)

As an example of a group we will take the set of all integers (denoted $\mathbb{Z}$) and the operation of addition. We can clearly see that any two integers added together will produce a third integer and so we satisfy the first condition. Also we know that addition of integers is associative, so we satisfy the second condition. We can take zero as an identity element because any integer added with zero is just the integer, and we meet the third condition. Lastly we need to find inverses for all elements. With the integers it is easy find, we just take the negative of any integer. So the last condition is met.
2.3.2 Rings

The concept of a ring can be viewed as an extension of the concept of a group. Like a group a ring is either an infinite set of elements of a finite set. A nonempty set \( R \) is said to be a ring if in \( R \) there are two operations, \(+\) and \( \ast \), such that:

\[
\begin{align*}
& a, b \in R \rightarrow a + b \in R \quad \text{(closed under addition)} \\
& a, b \in R \rightarrow a + b = b + a \quad \text{(\( R \) is abelian under addition)} \\
& a, b, c \in R \rightarrow (a + b) + c = a + (b + c) \quad \text{(associative under addition)} \\
& \exists 0 \in R : a + 0 = 0 + a = a, \forall a \in R \quad \text{(identity of addition exists)} \\
& \forall a \in R, \exists b \in R : a + b = b + a = 0 \quad \text{(inverse with addition exists)} \\
& a, b \in R \rightarrow a \ast b \in R \quad \text{(closed under multiplication)} \\
& a, b, c \in R \rightarrow a \ast (b \ast c) = (a \ast b) \ast c \quad \text{(associative under multiplication)} \\
& a, b, c \in R \rightarrow a \ast (b + c) = a \ast b + a \ast c \land (b + c) \ast a = b \ast a + c \ast a \quad \text{(Distributive law holds)}
\end{align*}
\]

As an example of a field we will take the set of real numbers (denoted \( R \)). The addition of two real numbers will give us a third real number, so \( R \) is closed under addition. The addition of two real numbers is abelian. Addition is associative when dealing with real numbers. We can take 0 as our identity element under addition. We can take the negative of any real number as its additive inverse. Multiplying two real numbers will yield a third real number. So \( R \) is closed under multiplication. Multiplication of real numbers is associative. And lastly the distributive law holds on real numbers. So \( R \) is a ring.
2.3.3 Fields

A field can be thought of as a special type of Ring. It follows all the condition of being a Ring with the added properties:

\[ a, b \in F, a \cdot b = 0 \rightarrow a = 0 \lor b = 0 \text{ (integral domain)} \]

\[ a \in F \land a \neq 0 \rightarrow \exists b \in F : a \cdot b = b \cdot a = 1 \text{ (Multiplicative inverse)} \]

As an example of a field we will again take the set of real numbers (R). We have already shown that R is a ring, so we will now explore the final two field properties. The only way for two numbers in R to multiply to zero is if one of the two numbers is zero. No other possibility exists. So R is an integral domain. Lastly if \( x \in R \) then its inverse is easy to find:

\[ x \in R \rightarrow (1/x) \in R \]
\[ x \cdot (1/x) = 1 \rightarrow x^{-1} = 1/x \]

So the last condition is met and R is indeed a field.

For AES we will be examining the finite field that is the set of integers modulo 2 (denoted \( Z_2 \)). This set contains the following elements: \{0,1\}. This is the same set used as the alphabet for bit strings. Addition within this field is simply a logical xor.

Expanding upon the concept of a field, we now introduce a Galois Field. Let \( p \) be a prime number then there exists a field \( GF(p^a) \) where \( a \) is a positive integer. This field is the set of all polynomials with degree less than \( a \), and coefficients in \( Z_p \). \( P \) is called the characteristic of the field. We will sometimes call Galois Fields simply finite fields of characteristic \( p \). Addition in this field is polynomial addition but the coefficients obey the rules of \( Z_p \). For our purposes \( P \) will always be 2. We will now explore some
examples in $GF(2^8)$. In this field the coefficients are either 0 or 1 (exist in $\mathbb{Z}_2$). So $x^6 + x^5 + x^2 + x + 1 \in GF(2^8)$. We will use bit strings to represent elements of this field in the following way:

$$\{0,1,1,0,0,1,1,1\} = x^6 + x^5 + x^2 + x + 1$$

The bits strings will always be of length 8, where the first bit represents $x^7$, the second bit represents $x^6$, and so on up until the last bit which represents 1. When adding elements of this field, we can simply apply a logical xor. The following equations are equivalent:

$$(x^7 + x^3 + x^2 + 1) + (x^5 + x^4 + x^2) = x^7 + x^5 + x^4 + x^3 + 1$$
$${1,0,0,0,1,1,0,1} \oplus {0,0,1,1,0,1,0,0} = {1,0,1,1,1,0,0,1}$$

We will sometimes use the hexadecimal shorthand to represent elements of this field.

The following equation is equivalent to the two previous.

$$8D \oplus 34 = B9$$

Multiplication in this field corresponds to multiplication of polynomials modulo an irreducible polynomial of degree 8. For AES we will use the following polynomial:

$$M(x) = \begin{cases} x^8 + x^4 + x^3 + 1 & (polynomial \ notation) \\ \{1,0,0,0,1,1,0,1\} & (bit \ string \ notation) \\ 11B & (Hex \ notation) \end{cases}$$

Let’s explore an example:

$$(x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

This gives us a polynomial not in our field, but we then apply the modulus:

$$(x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1) \mod (x^8 + x^4 + x^3 + 1) = x^7 + x^6 + 1$$

This result does exist within our field.
3 CryptKeeper

CryptKeeper is a visual programming environment designed to make the assembly and analysis of cryptographic ciphers easier. It is designed to be a supplementary tool to prospecting cryptographers. CryptKeeper hides much of the programming complexities from its users, and handles much of the mathematics required by modern ciphers. It allows users to visually watch the execution of ciphers and analyze how they process data. CryptKeeper allows users to focus more on the details of a cipher rather than the details of its implementation.

CryptKeeper is a node based visual programming environment. In CryptKeeper, users manipulate and link code nodes together. Code nodes are individual blocks that perform some operation. These nodes can take input data, and subsequently drive output data. The links between these nodes represent the flow of data from one node to another. Programs built in CryptKeeper will result in a structure similar to a directed graph. This structure provides the user with a visual outline of the system and its control flow.

3.1 User Interface

The CryptKeeper Graphical User Interface (GUI) consists of 4 separate windows. Users are free to relocate and resize these widows to their liking. These changes will persist between executions of the CryptKeeper program. We will now briefly explore the user interface.
3.1.1 CryptKeeper Window

The CryptKeeper window is the main window of the CryptKeeper system. Here users have access to the module canvas where they will create and edit their modules. This window also contains the controls for executing modules. Figure 3-1 shows the CryptKeeper window.

![Figure 3-1 The CryptKeeper Window](image)

Users have the option of visually running their module or running it in the background. A Visual run requires more overhead and thus will be slower than a background run. However during a visual run users have the option to pause and resume the modules execution at will. After a module has been paused, users can single step through the rest of a modules execution. Users also have the option of killing a running module at any time. A module will maintain its state during and after execution. During
a visual run the currently active node of the module will highlight green. Signals will highlight magenta when they contain bit string values. The visual execution will proceed one step at a time. Taking values from signals, processing these values and driving values are all considered individual steps. In between steps the system will pause for a brief amount of time. This time can be specified using the time delay field.

Creation and editing of modules takes place in the module canvas. Users select a Code Node type from the palette window and then click on the module canvas to place a new instance of the selected node type in the module canvas. If the edit option is selected in the palette window, users can relocate existing nodes to desired locations. Right clicking on a Code node will give users a menu option that will allow them to delete the node. Right clicking on a node’s ports will give users the option to connect a signal to that port.

3.1.2 Palette

The Palette window contains all the various types of Code Nodes. Users simply select the code node type they wish to add to the module canvas the click within the module canvas to add a new instance of the selected node type. Figure 3-2 shows the Palette window.
The Palette window separates nodes into various categories. These categories are: Bit Logic, Arithmetic, I/O, and Misc. Each category is given its own tab in the window. Within each tab is an edit option. This allows users to edit existing nodes within the current module canvas.

### 3.1.3 Property Viewer

The property viewer window displays various properties about the currently selected element in the module canvas. Figure 3-3 shows the Property Viewer window.
Selectable elements include Code Nodes and Signals. Some selectable elements will have editable properties and these will appear in the Editable Property section of this window. To edit these properties user can select the desired editable property then press the edit button. This will open a property specific editing window.

### 3.1.4 Console

The console window displays output from an executing module. Figure 3-4 shows the Console window.

![Console window](image)

There are separate tabs for both standard in and standard error.

### 3.2 CryptKeeper Programs

There are four basic elements to a CryptKeeper program: bit strings, code nodes, signals, and modules. Bit strings are the one data type available in CryptKeeper. Code Nodes are the work horses of a CryptKeeper program. They take in bit strings from signals and process this data in various ways. Some Code Nodes will produce new bit strings that are then pass on to and then processed by other Code Nodes. Signals are the links between code nodes. They transport outputs from one code node to another. A module is simply a collection of code nodes and the signals connecting these nodes.
3.2.1 Code Nodes

Code nodes are the basic operation block in CryptKeeper. Any work done by a program is done inside a code node. Code nodes contain various types of port groups. Port Groups can either be input groups or output groups. Input groups appear on top of a code node while output groups appear on the bottom. As their names suggest, input port groups accept incoming data, while output port groups drive outgoing data.

Both input groups and output groups will consist of one or more ports. Input groups contain input ports, and Output groups contain output ports. A code node will enter the execution queue when all the ports of one of its input groups have incoming data. Depending on which input group causes the node to be put in the queue, the output of node may vary. It should be noted that not all code nodes will have both input groups and output groups.

3.2.1.1 Bit Logic

The bit logic code nodes perform bitwise operations on bit strings. We will now explore the nodes in this category:

- XOr, And and Or
  - These nodes take in multiple bit string values (number defined by the user) and produces a bit string that is the result of iterating the given operation on the inputs starting from the left most input value to the right most input value. The operation is performed on the first two inputs, and then the result is paired with the third and so on (also know as a left fold).

- Not
The Not node takes in a single bit string value and outputs a new bit string whose value is the bitwise not of the input value.

**Concat**

- The Concat node takes in multiple bit strings and outputs a new bit string whose value is the concatenation of the inputs. Values are left folded.

**Permute**

- The Permute node takes in a single bit string value and produces an output bit string whose value is determined by the nodes permutation list. The list specifies which bits to extract from the input and the order they are to appear in resulting bit string. The resulting value may be wider or narrower than the input. The permutation list may contain duplicate values, in which case the same bit is used multiple times. It may also be a subset of the inputs available bits in which case the resulting value is narrower than the original.

**BitRange**

- The BitRange node takes in a single bit string value and produces a bit string whose value is equal to a subset of bits of the passed in value. The range of bits extracted is defined by the user.

**RotateLeft**

- The rotate left node takes in a bit string value and a second value that specifies the amount by which to rotate the first value. The bits of first value are rotated to the left. It then outputs this rotated value.

**RotateRight**
The rotate right node takes in a bit string value and a second value that specifies the amount by which to rotate the first value. The bits of first value are rotated to the right. It then outputs this rotated value.

- ShiftLeft
  - The shift left node takes in a bit string value and a second value that specifies the amount by which to shift the first value. The bits of the first value are shifted to the left. It then outputs this shifted value.

- ShiftRight
  - The shift right node takes in a bit string value and a second value that specifies the amount by which to shift right the first value. The bits of the first value are shifted to the right. It then outputs this shifted value.

- Partition
  - The partition takes in a single bit string value and will break up the value into equal sized parts. The size of the parts depends on the width of the passed in bit string and the number of outputs specified by the user.

- ConstantRotateLeft
  - The constant rotate left node takes in a bit string value and rotates the bits of that value to the left by a user specified amount. It then outputs this rotated value.

- ConstantRotateRight
  - The constant rotate right node takes in a bit string value and rotates the bits of that value to the right by a user specified amount. It then outputs this rotated value.
3.2.1.2 Arithmetic

Arithmetic nodes are used to perform arithmetic operations on bit strings.

- **Add and Multiply**
  - These nodes take in multiple bit string values (number defined by the user) and produces a bit string that is the result of left folding the operation on the input values. These operations will never cause overflow.

- **Subtract and Divide**
  - These nodes take in multiple bit string values (number defined by the user) and produces a bit string that is the result of left folding the operation on the input values. The result of a subtraction will never be negative. Any values that should produce a negative value will actually produce a value of zero.

- **Divide**
  - The divide node takes in multiple bit string values (number defined by the user) and produces a bit string that is the result of left folding the operation on the input values. The result of a divide operation is always rounded down to the closest integer value.

- **Modulo**
  - The modulo nodes take in multiple bit string values (number defined by the user) and produces a bit string that is the result of left folding the operation on the input values.

- **Power**
• The power node takes two bit string values and raises the first bit string to the power of the second bit string.

• RestrictedAdd
  o The RestrictedAdd node takes in multiple bit string values (number defined by the user) and produces a bit string that is the result of left folding the operation on the input values. The results of a restricted add operation will be truncated down into a value that contains a user specified number of bits.

3.2.1.3 Input/Output

• Print
  o A Print node accepts one input and upon execution, will print the value of the received bit string to the console window. This value will be printed in hexadecimal.

• Constant, ConstantMatrix, FF2Constant, FF2ConstantMatrix
  o These nodes will drive a user specified constant value. These nodes will drive this value only once at the start of a modules execution. The Constant node drives a single bit string value, ConstantMatrix drives a matrix of bit strings, FF2Constant will drive a bit string that exists in a user specified finite field of characteristic 2, and the FF2ConstantMatrix drives a matrix of these values.

• Triggered Constant, TriggeredConstantMatrix, TriggeredFF2Constant, and TriggeredFF2Matrix
- These nodes take a single input. The actual value of this input is irrelevant. It is simply used as a trigger telling the node to drive its user defined value.

- **DifferentialSet**
  - A DifferentialSet node will drive a user defined set of bit string values. The set also keeps track of the differential of all the values in the set. The differential is simply the value of the xor of all values in the set. This is used to simulating passing multiple values through a system simultaneously while keeping track of those values differential.

- **LambdaSet**
  - A LambdaSet node drives a set of 256 FF2Matrix values. The values of the cells of these matrices are defined by the user. Cells can be either active or constant. Constant cells are simply a constant value that is the same across all 256 matrices. The values of active cells vary throughout the 256 matrices, taking on the values 0-255 (in decimal). In addition to the 256 FF2Matrix values, the set keeps track of the differential of all the matrices. The differential is simply the xor of all the matrices.

- **IndexedConstantList**
  - The IndexedConstantList node contains a user defined list of bit strings. It contains a single input port that takes in an index value. When it receives an index value it drives the bit string value in the list at the specified index. Indexing out of bounds is an error.
3.2.1.4 Miscellaneous

- **ModuleNode**
  - A ModuleNode is an interface used to nest one module inside another. The ModuleNodes inputs and outputs will correspond to those defined in the module enclosed within the node. Users specify what module the node encloses.

- **Repeater**
  - A Repeater node takes a single input value and then repeats that value on one or more output ports. The number of times this value is repeated is determined by the user.

- **SBox**
  - A SBox (Substitution Box) node contains a user defined lookup table of bit string values. It takes in a single bit string value then parses the bits of that value to determine a row and column index. The bits used to determine the row and column indices are determined by the user. Once a row and column have been determined, the SBox drives the specified value.

- **Join**
  - A Join node is used to combine multiple data paths into one path. The Join node has multiple input groups each with a single input port. When any one of the ports receives a value, the Join node will simply forward the received value on its single output port.

- **Loop**
A Loop node is used, not surprisingly, to effect loops in a module. The user specifies the number of times the Loop node should repeat. The node has two input groups each with a single input port. One of these groups is used to start the looping. It will cause the node to initialize itself and then forward out the received data along with a counter value that signifies the iteration count of the loop. The second input group is used to feed back data to the loop node and cause the next iteration to begin. Once all the iterations have completed, the node will drive its last received value out its “loop end” port.

BitStringList

A BitStringList node is used to store Bit String values generated at runtime. It has two input groups, one to add new values, and one to request value using their index within the list. When a value is added to the list, no output value is produced. When a value is requested on the index port, it is driven out the nodes only output port.

FlattenMatrix

The FlattenMatrix node takes in a matrix value and produces a bit strings whose value is the concatenation of all the cells of the matrix. Concatenation starts with the first cell of the first row, then concatenates cell 2 of the first row, then cell 3 and so on. It proceeds row by row until all values have been concatenated.

ToMatrix
o The ToMatrix node takes a bit string value and transforms it into a bit string matrix. The dimensions of the matrix are defined by the user. When the node receives a bit string value, it first partitions the value into equal parts. The number of partitions is equal to the number of rows of the matrix. Then each of these partitions is again partitioned into equal parts, one for each column.

- TransposeMatrix
  o The TransposeMatrix node takes in a bit string matrix and outputs a bit string matrix that is a transposition of the received value.

### 3.2.2 Signals

Signals represent data paths within a module. Signals connect output ports to input ports. Signals carry the one data type allowed in CryptKeeper, the bit string. Signals keep a history of all values that have been sent over them. The signals history and any currently held value can be seen in the property viewer.

### 3.2.3 Modules

A module is simply a set of code nodes, and their associated signals that link the nodes to each other. A module may also contain two special nodes that handle taking inputs from external sources and sending output data to external sources. These nodes are named External Inputs and External outputs. Having these nodes allows a module to be nested within another module.
3.2.4 Program Execution

CryptKeeper has an engine that controls the execution of a module. The engine keeps an execution queue (First in First out) which is used to control the order of code node execution. When a module begins its execution, its code nodes are search for any nodes that are simply constant nodes. Constant nodes are those that simply contain a constant value. These nodes are then put into the execution queue. These value will be driven only once at the start of a modules execution. The path this initial data takes determines the resulting operation of the module. As the program executes, data will pass from one code node to another. When a code node has an input group with data waiting on all of the group’s input ports, it will be added to the execution queue. Execution will continue to occur until the engine finds the execution queue to be empty. At this time execution is said to be completed.

3.3 Example

In computer science, when learning a new programming language, it is customary to make the first program taught be a “Hello World” program. A “Hello World” program simply prints out the character string “Hello World” to a display device. CryptKeeper doesn’t have character strings as a data type so instead we will print a bit string value. Bit string values are printed to the console using their hexadecimal representation. Figure 3-4 shows the structure of a hello value program.
The program consists of two code nodes and a single signal connecting them. The constant node drives its value and the print node prints the value to the console.
4 DES Assembly and Analysis

The Data Encryption Standard (DES) is a symmetric-key block cipher [6]. It was selected as the Federal Information Processing Standard (FIPS) for the United States in 1976. DES has a 64-bit block size, and a 64-bit key. The key is presented as a 64-bit value, but 8 of these bits are used to check parity so the key size of DES is effectively 56-bits. We will now explore the structure of DES. A common approach to teaching DES is to show diagrams that depicts the various operations and data flow of the algorithm. It was this very approach to teaching DES that gave birth to the idea for a visual programming environment for cryptographic ciphers.

In this section we will build the DES algorithm from the ground up. Starting with its internals and expanding outward. The diagrams presented in this section (and those that follow) will be the actual implementation of the algorithm in the CryptKeeper system.

4.1 The DES F Function

First we will explore the F function. The structure of the F function is depicted in figure 4-1.
The F function (or module in CryptKeeper) takes two inputs, where the first input is a 32-bit value and the second is a 48-bit value. We will refer to the first input as R and the second input as the RoundKey. When the function starts R is passed to the E node. The E node is a permute node which produces a 48-bit output. The permutation performed by E is defined in table 4-1.

<table>
<thead>
<tr>
<th>E Permutation Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 0 1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>7 8 9 10 11 12 13 14 15 16</td>
</tr>
<tr>
<td>15 16 17 18 19 20 21 22 23 24</td>
</tr>
<tr>
<td>23 24 25 26 27 28 29 30 31 0</td>
</tr>
</tbody>
</table>

Table 4-1 E Permutation Table
So the first bit of $E(R)$ is the $31^{st}$ bit of $R$, the second bit of $E(R)$ is the $0^{th}$ bit of $R$ and so on. Since $R$ is only 32-bits and $E(R)$ is 48-bits, we have some repetition of bits. $E$ is said to expand $R$. The output of $E$ is then xor-ed with the RoundKey and the resulting value is passed to the partition node. The partition node separates the input value into eight 6-bit values. The first 6-bit value contains bits 0-5, the second contains bits 6-11 and so on. These values are passed to S1 thru S8 respectively. S1 thru S8 are SBox nodes that take a 6-bit value and output a 4-bit value. These SBox nodes substitute the 6-bit value passed in with a 4-bit value that is chosen from the SBox nodes lookup table. They use bits 0 and 5 to determine a row, and bits 1,2,3, and 4 to determine a column. The lookup table for S1 is defined in table 4-2.

<table>
<thead>
<tr>
<th>S1</th>
<th>Row\Col</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>4</td>
<td>13</td>
<td>1</td>
<td>2</td>
<td>15</td>
<td>11</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td>6</td>
<td>12</td>
<td>5</td>
<td>9</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>15</td>
<td>7</td>
<td>4</td>
<td>14</td>
<td>2</td>
<td>13</td>
<td>1</td>
<td>10</td>
<td>6</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>14</td>
<td>8</td>
<td>13</td>
<td>6</td>
<td>2</td>
<td>11</td>
<td>15</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>12</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>11</td>
<td>3</td>
<td>14</td>
<td>10</td>
<td>0</td>
<td>6</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-2 Lookup table for S1

The nodes S2 thru S8 operate in a similar fashion but use their own unique look up tables (See Appendix A for these tables). The outputs of S1 thru S8 are concatenated together to generate a 32-bit value. This value is then passed to the P node which is a permute node. The P node permutes the input value in a manner similar to the E node but uses the values in table 4-3.

<table>
<thead>
<tr>
<th>P Permutation Table</th>
<th>15</th>
<th>6</th>
<th>19</th>
<th>20</th>
<th>28</th>
<th>11</th>
<th>27</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>22</td>
<td>25</td>
<td>4</td>
<td>17</td>
<td>30</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>23</td>
<td>13</td>
<td>31</td>
<td>26</td>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>29</td>
<td>5</td>
<td>21</td>
<td>10</td>
<td>3</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-3 The P Permutation Table

The result of the P function becomes the output of the F function.
4.2 The DES Round

Next we will examine the DES round function. A diagram depicting the DES round function can be seen in figure 4-2.

The DES round function takes in two input values. The first is a 64-bit value that we will call Data, and the second is a 48-bit value that we will call the RoundKey. The DES round function subsequently partitions Data value input into two 32-bit values L0 and R0. R0 and the RoundKey are passed to the F function. The result of F is a 32-bit value, and this value is xor-ed with L0. The result of this xor becomes R1. L1 is simply a copy.
of R0. Then L1 is concatenated with R1 to produce a 64-bit value. This 64-bit value becomes the output of the round.

### 4.3 DES Algorithm

Finally we will look at the Entire DES algorithm. The DES algorithm is depicted in figure 4-3.

![Figure 4-3 The Full DES Algorithm](image)

The full DES algorithm requires two pieces of information, a 64-bit plain text, and a 64-bit master key. The DES algorithm proceeds as follows. The Master key is given to the KeySchedule which is a module node. This module generates and internally stores the
various round keys. For brevity we will not explore the internals of the key schedule.

After the round keys have been generated by the KeySchedule node, the plain text value
is fed to the IP node. IP is a permute node that permutes the plain text using the values in
table 4-4.

<table>
<thead>
<tr>
<th>IP Permutation Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>57 49 41 33 25 17 9 1</td>
</tr>
<tr>
<td>59 51 43 35 27 19 11 3</td>
</tr>
<tr>
<td>61 53 45 37 29 21 13 5</td>
</tr>
<tr>
<td>63 55 47 39 31 23 15 7</td>
</tr>
<tr>
<td>65 48 40 32 24 16 8 0</td>
</tr>
<tr>
<td>67 50 42 34 26 18 10 2</td>
</tr>
<tr>
<td>69 52 44 36 28 20 12 4</td>
</tr>
<tr>
<td>71 54 46 38 30 22 14 6</td>
</tr>
</tbody>
</table>

Table 4-4 IP Permutation Table

This permuted value is fed to the Loop node which will forward the data to the DES
round node and request the appropriate key from the KeySchedule node be given to the
DES round node. The loop note will iterate this process 16 times. After all 16 rounds
have been processed, the output is passed through the FP permute node. The FP node
uses the values in table 4-5.

<table>
<thead>
<tr>
<th>FP Permutation Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>39 7 47 15 55 23 63 31</td>
</tr>
<tr>
<td>38 6 46 14 54 22 62 30</td>
</tr>
<tr>
<td>37 5 45 13 53 21 61 29</td>
</tr>
<tr>
<td>36 4 44 12 52 20 60 28</td>
</tr>
<tr>
<td>35 3 43 11 51 19 59 27</td>
</tr>
<tr>
<td>34 2 42 10 50 18 58 26</td>
</tr>
<tr>
<td>33 1 41 9 49 17 57 25</td>
</tr>
<tr>
<td>32 0 40 8 48 16 56 24</td>
</tr>
</tbody>
</table>

Table 4-5 FP Permutation Table

The permutation effected by FP is the inverse of the permutation effected by IP. The
output of the FP node is the DES algorithms resulting cipher text.
5 AES Assembly and Analysis

AES is a standard based on the Rijndael algorithm, a symmetric block cipher that can process data blocks of 128 bits, using a key of sizes 128, 192, and 256 [7]. For the purposes of this paper, we will only concern ourselves with a key size of 128. We will now explore the structure of the AES algorithm. As with the DES algorithm we will examine the structure of AES from the bottom up. We will define its inner most pieces first then see how those pieces fit together.

5.1 The State

The AES algorithm operates on a two-dimensional matrix of bytes called the State. The State consists of a 4x4 matrix. Elements of the State matrix are bit strings that exist in $GF(2^8)$. The State in CryptKeeper is implemented using a FF2Matrix node. The FF2 matrix is of degree 8, and uses the bit string “0x11b” as the irreducible polynomial. Let $S$ be the state, then we will use the following notation to represent the State:

$$S = \begin{bmatrix}
S_{0,0}, & S_{0,1}, & S_{0,2}, & S_{0,3} \\
S_{1,0}, & S_{1,1}, & S_{1,2}, & S_{1,3} \\
S_{2,0}, & S_{2,1}, & S_{2,2}, & S_{2,3} \\
S_{3,0}, & S_{3,1}, & S_{3,2}, & S_{3,3}
\end{bmatrix}$$

5.2 SubBytes

The SubBytes transformation is a non-linear substitution operation that acts on each byte of the State individually. Each byte transformation is defined mathematically, but for the purposes of this paper we will ignore this detail. We will simply think of the
SubBytes function as a predefined lookup table that is applied to each byte of the state.

Table 5-1 shows the lookup table used by SubBytes.

<table>
<thead>
<tr>
<th>Row\Column</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63</td>
<td>7c</td>
<td>77</td>
<td>7b</td>
<td>f2</td>
<td>6b</td>
<td>6f</td>
<td>c5</td>
<td>30</td>
<td>01</td>
<td>67</td>
<td>2b</td>
<td>fe</td>
<td>D7</td>
<td>ab</td>
<td>76</td>
</tr>
<tr>
<td>1</td>
<td>82</td>
<td>C9</td>
<td>7d</td>
<td>fa</td>
<td>59</td>
<td>47</td>
<td>f0</td>
<td>ad</td>
<td>d4</td>
<td>a2</td>
<td>af</td>
<td>9c</td>
<td>a4</td>
<td>72</td>
<td>c0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b7</td>
<td>fd</td>
<td>93</td>
<td>26</td>
<td>36</td>
<td>3f</td>
<td>f7</td>
<td>cc</td>
<td>34</td>
<td>a5</td>
<td>e5</td>
<td>f1</td>
<td>71</td>
<td>d8</td>
<td>31</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>04</td>
<td>c7</td>
<td>23</td>
<td>c3</td>
<td>18</td>
<td>96</td>
<td>05</td>
<td>9a</td>
<td>07</td>
<td>12</td>
<td>80</td>
<td>e2</td>
<td>eb</td>
<td>27</td>
<td>b2</td>
<td>75</td>
</tr>
<tr>
<td>4</td>
<td>09</td>
<td>83</td>
<td>2c</td>
<td>1a</td>
<td>1b</td>
<td>6e</td>
<td>5a</td>
<td>a0</td>
<td>52</td>
<td>3b</td>
<td>d6</td>
<td>b3</td>
<td>29</td>
<td>e3</td>
<td>2f</td>
<td>84</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>d1</td>
<td>00</td>
<td>ed</td>
<td>20</td>
<td>fc</td>
<td>b1</td>
<td>5b</td>
<td>6a</td>
<td>cb</td>
<td>be</td>
<td>39</td>
<td>4a</td>
<td>4c</td>
<td>58</td>
<td>cf</td>
</tr>
<tr>
<td>6</td>
<td>D0</td>
<td>ef</td>
<td>aa</td>
<td>fb</td>
<td>43</td>
<td>4d</td>
<td>33</td>
<td>85</td>
<td>45</td>
<td>f9</td>
<td>02</td>
<td>7f</td>
<td>50</td>
<td>3c</td>
<td>9f</td>
<td>a8</td>
</tr>
<tr>
<td>7</td>
<td>51</td>
<td>a3</td>
<td>40</td>
<td>8f</td>
<td>92</td>
<td>9d</td>
<td>38</td>
<td>f5</td>
<td>bc</td>
<td>b6</td>
<td>da</td>
<td>21</td>
<td>10</td>
<td>ff</td>
<td>f3</td>
<td>d2</td>
</tr>
<tr>
<td>8</td>
<td>cd</td>
<td>0c</td>
<td>ec</td>
<td>5f</td>
<td>97</td>
<td>44</td>
<td>17</td>
<td>c4</td>
<td>a7</td>
<td>7e</td>
<td>3d</td>
<td>64</td>
<td>5d</td>
<td>19</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>60</td>
<td>81</td>
<td>4f</td>
<td>dc</td>
<td>22</td>
<td>2a</td>
<td>90</td>
<td>88</td>
<td>46</td>
<td>ee</td>
<td>b8</td>
<td>14</td>
<td>de</td>
<td>5e</td>
<td>0b</td>
<td>7b</td>
</tr>
<tr>
<td>A</td>
<td>e0</td>
<td>32</td>
<td>3a</td>
<td>0a</td>
<td>49</td>
<td>06</td>
<td>24</td>
<td>5c</td>
<td>c2</td>
<td>d3</td>
<td>ac</td>
<td>62</td>
<td>91</td>
<td>95</td>
<td>e4</td>
<td>79</td>
</tr>
<tr>
<td>B</td>
<td>e7</td>
<td>c8</td>
<td>37</td>
<td>6d</td>
<td>8d</td>
<td>D5</td>
<td>4e</td>
<td>a9</td>
<td>6c</td>
<td>56</td>
<td>f4</td>
<td>ea</td>
<td>65</td>
<td>7a</td>
<td>ae</td>
<td>08</td>
</tr>
<tr>
<td>C</td>
<td>ba</td>
<td>78</td>
<td>25</td>
<td>2e</td>
<td>1c</td>
<td>A6</td>
<td>b4</td>
<td>c6</td>
<td>e8</td>
<td>dd</td>
<td>74</td>
<td>1f</td>
<td>4b</td>
<td>bd</td>
<td>8b</td>
<td>8a</td>
</tr>
<tr>
<td>D</td>
<td>70</td>
<td>3e</td>
<td>b5</td>
<td>66</td>
<td>48</td>
<td>03</td>
<td>f6</td>
<td>0e</td>
<td>61</td>
<td>35</td>
<td>57</td>
<td>b9</td>
<td>86</td>
<td>c1</td>
<td>1d</td>
<td>9e</td>
</tr>
<tr>
<td>E</td>
<td>e1</td>
<td>f8</td>
<td>98</td>
<td>11</td>
<td>69</td>
<td>d9</td>
<td>8e</td>
<td>94</td>
<td>9b</td>
<td>1e</td>
<td>87</td>
<td>e9</td>
<td>ce</td>
<td>55</td>
<td>28</td>
<td>df</td>
</tr>
<tr>
<td>F</td>
<td>8d</td>
<td>a1</td>
<td>89</td>
<td>0d</td>
<td>bf</td>
<td>e6</td>
<td>42</td>
<td>68</td>
<td>41</td>
<td>99</td>
<td>2d</td>
<td>0f</td>
<td>b0</td>
<td>54</td>
<td>bb</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 5-1 SubBytes Lookup Table

In CryptKeeper SubBytes is implemented using a SBox node that uses the values in table 5-1 for its lookup table. The node takes an 8-bit value and uses bits 0, 1, 2, and 3 to determine a row, and bits 4, 5, 6, and 7 to determine a column.

### 5.3 ShiftRow

In the ShiftRow transformation, the bytes of each row of our State matrix are rotated left by different amounts. The first row is rotated by 0, second by 1, third by 2, and the forth by 3. Example:

\[
\text{ShiftRow} \begin{bmatrix}
S_{0,0}, S_{0,1}, S_{0,2}, S_{0,3} \\
S_{1,0}, S_{1,1}, S_{1,2}, S_{1,3} \\
S_{2,0}, S_{2,1}, S_{2,2}, S_{2,3} \\
S_{3,0}, S_{3,1}, S_{3,2}, S_{3,3}
\end{bmatrix} = \begin{bmatrix}
S_{0,0}, S_{0,1}, S_{0,2}, S_{0,3} \\
S_{1,1}, S_{1,2}, S_{1,3}, S_{1,0} \\
S_{2,2}, S_{2,3}, S_{2,0}, S_{2,1} \\
S_{3,3}, S_{3,0}, S_{3,1}, S_{3,2}
\end{bmatrix}
\]

In CryptKeeper this is implemented as a permute node using the values in table 5-2.
5.4 MikColumn

The MixColumn function operates on the State matrix column by column. For the purposes of this paper we look at this function as simply a matrix multiplication. A fixed matrix is multiplied by the state and the result is the new state.

\[
\begin{bmatrix}
S_{0,0}, S_{0,1}, S_{0,2}, S_{0,3} \\
S_{1,0}, S_{1,1}, S_{1,2}, S_{1,3} \\
S_{2,0}, S_{2,1}, S_{2,2}, S_{2,3} \\
S_{3,0}, S_{3,1}, S_{3,2}, S_{3,3}
\end{bmatrix}
\times
\begin{bmatrix}
02, 03, 01, 01 \\
01, 02, 03, 01 \\
01, 02, 03, 03 \\
03, 01, 01, 02
\end{bmatrix}
= \begin{bmatrix}
S_{0,0}, S_{0,1}, S_{0,2}, S_{0,3} \\
S_{1,0}, S_{1,1}, S_{1,2}, S_{1,3} \\
S_{2,0}, S_{2,1}, S_{2,2}, S_{2,3} \\
S_{3,0}, S_{3,1}, S_{3,2}, S_{3,3}
\end{bmatrix}
\]

We will call this fixed matrix A. In CryptKeeper A is implemented using a TriggeredFF2Matrix. MixColumn is implemented by simply multiplying A by the State.

5.5 AddRoundKey

The round key in AES is represented as a 4x4 matrix of bytes. In this step the round key is simply added to the state. The resulting matrix is the new State. Remember in this context addition is the xor operation. In CryptKeeper this is implemented using a xor node. (We could also have using and add node, being that addition in the specified field is xor)

5.6 AES Round

Next we will examine a round in the AES algorithm. Figure 5-1 depicts the AES round function.
The AES Round takes in two matrices as its inputs. The first is the State and the second is the round key. The State is passed to the SubBytes SBox node whose result is forwarded to the ShiftRows permute node. The result of the ShiftRow node is passed to a repeater that will repeat the value twice. The first copy is passed to the A node which will cause the A matrix to be passed to the MixColumn node. The second value from the repeater node is also passed to the MixColumn node. The MixColumn node is just a
multiply node. The result of this multiply is then xor-ed with the round key. The resulting value is the output of the AES Round.

### 5.7 AES Final Round

The final round of AES is similar to the regular round of AES save for it excludes the MixColumn function. Figure 5-1 depicts its implementation in CryptKeeper.

![Figure 5-2 AES Final Round](image)

### 5.8 AES Algorithm

We will now explore the AES algorithm. Figure 5-3 shows a CryptKeeper implementation of 3 rounds of AES
We present a 3 round version here rather than the full 10 round version for brevity. The 10 round version of AES follows a similar pattern. The key point to remember is that the final round of AES differs from its preceding rounds. The flow of data is easy to follow in this example. The plaintext is first xor-ed with the master key then fed to the first round of AES along with the round 1 key. Subsequent rounds take their preceding rounds output as input along with their respective round key. The round keys are derived from the master key by the key schedule, but for brevity, we do not show the key schedule.
6 Salsa20 Assembly and Analysis

The Salsa20 encryption function is a modern stream cipher developed by Daniel J Bernstein [8]. The heart of the Salsa20 stream cipher is the Salsa20 hash function. As with our previous two examples we will build Salsa20 in a ground up fashion. First we define the Salsa20 hash function, then the Salsa expansion function and finally the salsa20 encryption function.

6.1 Quarter Round

First we will look at the quarterround function:

Let \( y = (y_0, y_1, y_2, y_3) \), where \( y_i \) is a 4-byte value
Let \( z = (z_0, z_1, z_2, z_3) \), where
\[
\begin{align*}
  z_1 &= y_1 \oplus ((y_0 + y_3) \ll 7) \\
  z_2 &= y_2 \oplus ((z_1 + y_0) \ll 9) \\
  z_3 &= y_3 \oplus ((z_2 + z_1) \ll 13) \\
  z_0 &= y_0 \oplus ((z_3 + z_2) \ll 18)
\end{align*}
\]
then \( \text{quarterround}(y) = z \)

In quarter round addition is taken modulo \( 2^{32} \). Figure 6-1 shows quarterround’s implementation in CryptKeeper.
Figure 6-1 Quaterround

This may appear a bit daunting at first glance. However, visually executing this module servers to clarify this confusion. Each column of nodes in the module represents the equations defining $Z_0$ thru $Z_4$. Following the execution shows that the input is partitioned into four 4-byte values, Y0 thru Y3. Y0 thru Y3 are repeater nodes and help the columns to closely mimic the equations defining $Z_0$ thru $Z_4$. The Z0 thru Z4 nodes are also repeater nodes and will hold the values of $Z_0$ thru $Z_4$ respectively. When the Z1 column completes it triggers the Z2 column which in turn triggers the Z3 column, which finally triggers the Z0 column. After all the columns have executed, there output values are
concatenated together and this value becomes the modules output. The nodes named “+” are restricted add nodes (restricted to 32-bits). The nodes named “<<<#” are left rotation nodes (rotating by the value of #).

6.2 Row Round

Now we shall examine the rowround function.

Let

\[
\begin{bmatrix}
  y_0, y_1, y_2, y_3 \\
  y_4, y_5, y_6, y_7 \\
  y_8, y_9, y_{10}, y_{11} \\
  y_{12}, y_{13}, y_{14}, y_{15}
\end{bmatrix}
\]

, \quad z =

\[
\begin{bmatrix}
  z_0, z_1, z_2, z_3 \\
  z_4, z_5, z_6, z_7 \\
  z_8, z_9, z_{10}, z_{11} \\
  z_{12}, z_{13}, z_{14}, z_{15}
\end{bmatrix}
\]

where \( y \) and \( z \) are 4-byte values

\[
(z_0, z_1, z_2, z_3) = \text{quaterround}(y_0, y_1, y_2, y_3)
\]

\[
(z_4, z_5, z_6, z_7) = \text{quaterround}(y_4, y_5, y_6, y_7)
\]

\[
(z_8, z_9, z_{10}, z_{11}) = \text{quaterround}(y_8, y_9, y_{10}, y_{11})
\]

\[
(z_{12}, z_{13}, z_{14}, z_{15}) = \text{quaterround}(y_{12}, y_{13}, y_{14}, y_{15})
\]

then \( \text{rowround}(y) = z \)

So the input and output of the function is a byte matrix. Figure 6-2 shows the quaterround implementation in CryptKeeper.
If we look at the inputs and outputs of the quarterround functions within rowround, we see that the rows of the matrix are permuted prior to being passed to quarterround, and this permutation is inverted prior to being output from the module. So the input to the module is passed to a permute node which transforms the input matrix as follows:

\[
\begin{bmatrix}
y_0, y_1, y_2, y_3 \\
y_4, y_5, y_6, y_7 \\
y_8, y_9, y_{10}, y_{11} \\
y_{12}, y_{13}, y_{14}, y_{15}
\end{bmatrix}
\rightarrow
\begin{bmatrix}
y_0, y_1, y_2, y_3 \\
y_2, y_6, y_7, y_4 \\
y_{10}, y_{11}, y_8, y_9 \\
y_{15}, y_{12}, y_{13}, y_{14}
\end{bmatrix}
\]

The permutation is defined by table 6-1.
The permuted matrix is then flattened to a 64-byte scalar value, which is then partitioned to 4 16-byte values. These 4 values are fed to the quarterround functions. The outputs of the quarterround functions are concatenated back into a 64-byte value, and this value is turned back into a 4x4 matrix (each element is 4-bytes). The matrix has the permutation inverted and the value becomes the output of our module. The inversion of the initial permutation is defined in table 6-2.

<table>
<thead>
<tr>
<th>Rowround Initial Permutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

*Table 6-1 Rowrounds Permutation Table*

<table>
<thead>
<tr>
<th>Rowround Inverse Permutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>13</td>
</tr>
</tbody>
</table>

*Table 6-2 Rowround Inverse Permutation*

### 6.3 Column Round

Next let us examine the column round function.

Let \( y = \begin{bmatrix} y_0, y_1, y_2, y_3 \\ y_4, y_5, y_6, y_7 \\ y_8, y_9, y_{10}, y_{11} \\ y_{12}, y_{13}, y_{14}, y_{15} \end{bmatrix} \), then \( z = \begin{bmatrix} z_0, z_1, z_2, z_3 \\ z_4, z_5, z_6, z_7 \\ z_8, z_9, z_{10}, z_{11} \\ z_{12}, z_{13}, z_{14}, z_{15} \end{bmatrix} \)

where \( y_i \) and \( z_j \) are 4-byte values

\[
(z_0, z_4, z_8, z_{12}) = \text{quarterround}(y_0, y_4, y_8, y_{12})
\]

\[
(z_5, z_9, z_{13}, z_i) = \text{quarterround}(y_5, y_9, y_{13}, y_i)
\]

\[
(z_{10}, z_{14}, z_2, z_6) = \text{quarterround}(y_{10}, y_{14}, y_2, y_6)
\]

\[
(z_{15}, z_3, z_7, z_{11}) = \text{quarterround}(y_{15}, y_3, y_7, y_{11})
\]

then \( \text{columnround}(y) = z \)
If we examine the definition we see that this is similar to the rowround function only the matrix is transposed prior to entering the quarterround functions, then the transposition is inverted prior to being output from the module.

\[
\begin{pmatrix}
Y_0, Y_1, Y_2, Y_3 \\
Y_4, Y_5, Y_6, Y_7 \\
Y_8, Y_9, Y_{10}, Y_{11} \\
Y_{12}, Y_{13}, Y_{14}, Y_{15}
\end{pmatrix} = \begin{pmatrix}
Y_0, Y_1, Y_2, Y_3 \\
Y_4, Y_5, Y_6, Y_7 \\
Y_8, Y_9, Y_{10}, Y_{11} \\
Y_{12}, Y_{13}, Y_{14}, Y_{15}
\end{pmatrix}^T \end{pmatrix}^T
\]

Figure 6-3 shows column round defined in CryptKeeper.

The input matrix is transposed, passed to rowround, then the output has the transposition inverted.

### 6.4 Double Round

Next we examine double round.
Let $y = \begin{bmatrix} y_0, y_1, y_2, y_3 \\ y_4, y_5, y_6, y_7 \\ y_8, y_9, y_{10}, y_{11} \\ y_{12}, y_{13}, y_{14}, y_{15} \end{bmatrix}$, $z = \begin{bmatrix} z_0, z_1, z_2, z_3 \\ z_4, z_5, z_6, z_7 \\ z_8, z_9, z_{10}, z_{11} \\ z_{12}, z_{13}, z_{14}, z_{15} \end{bmatrix}$

where $y_i$ and $z_i$ are 4-byte values

then doubleround $(x) = $ rowround $($columnround $(x)) = z$

Figure 6-4 shows double round defined in CryptKeeper.

The input is fed to columnround whose output is passed to rowround and the result becomes the modules output.

### 6.5 Little Endian

Now we examine the little-endian function.

$let \ b be \ a \ 4\_byte \ value$

then $b = (b_0, b_1, b_2, b_3)$, where $b_i \ are \ 1\_byte \ values$

then littleendian $(b) = (b_3, b_2, b_1, b_0)$

Simply put, the little endian function takes a 4-byte and reverses the order of its bytes.

Figure 6-5 shows littleendian defined in CryptKeeper.
The 4-byte input is partitioned into 4 1-byte values, and then these values are concatenated back together in reverse order. This new value becomes the output.

### 6.6 Matrix Little Endian

Next we define matrixlittleendian which will take a 4x4 matrix of 4-byte values and perform little-endian on each of these values.

\[
\begin{pmatrix}
\text{littleendian}(y_0), & \text{littleendian}(y_1), & \text{littleendian}(y_2), & \text{littleendian}(y_3) \\
\text{littleendian}(y_4), & \text{littleendian}(y_5), & \text{littleendian}(y_6), & \text{littleendian}(y_7) \\
\text{littleendian}(y_8), & \text{littleendian}(y_9), & \text{littleendian}(y_{10}), & \text{littleendian}(y_{11}) \\
\text{littleendian}(y_{12}), & \text{littleendian}(y_{13}), & \text{littleendian}(y_{14}), & \text{littleendian}(y_{15})
\end{pmatrix}
\]

\[
\begin{align*}
\text{matrixlittleendian} &= \\
\begin{pmatrix}
y_0, & y_1, & y_2, & y_3 \\
y_4, & y_5, & y_6, & y_7 \\
y_8, & y_9, & y_{10}, & y_{11} \\
y_{12}, & y_{13}, & y_{14}, & y_{15}
\end{pmatrix}
\end{align*}
\]
Figure 6-6 shows matrixlittleendian defined in CryptKeeper.

![Diagram of Matrix Little Endian]

The input matrix is flattened into a 64-byte scalar value, which is then partitioned into 16 4-byte values, each of these values is passed thru the littleendian function. The results of the littleendian functions are concatenated back together and then turned back into a 4x4 matrix of 4-byte values.

### 6.7 Salsa20 Hash

Now we will look at the Salsa20 hash function.
Let $x = \begin{bmatrix} x_0, x_1, x_2, x_3 \\ x_4, x_5, x_6, x_7 \\ x_8, x_9, x_{10}, x_{11} \\ x_{12}, x_{13}, x_{14}, x_{15} \end{bmatrix}$

Let $y = \text{matrixlittleendian}(x)$

then $\text{Salsa20Hash}(x) = \text{matrixlittleendian}(y + \text{doubleround}^{10}(y))$

Figure 6-7 shows Salsa20Hash defined in CryptKeeper.
The input is passed to the matrixlittleendian function and is then passed to a repeater node. This node sends one copy of the value to the loop node, which will forward it on to the doubleround function. Doubleround then passes its output back to the loop function to complete one iteration. This looping iterates 10 times then the loop function sends the output of the last doubleround to the restricted add node. Here the value is added to the littleendian value of the input. The result of the addition is passed to the littleendian function its output becomes the output of the module.

### 6.8 Salsa20 Expansion

Next we will explore the Salsa20 expansion function. The Salsa20 expansion function has 2 variants, one where k is a 32-byte value, and one where k is a 16 byte value. We will only concern ourselves with the case where k is a 32-byte value.

Define:

\[
\begin{align*}
\sigma_0 &= (101, 120, 112, 97), \sigma_1 = (110, 100, 32, 51), \\
\sigma_2 &= (50, 45, 98, 121), \sigma_3 = (116, 101, 32, 107) \\
k &= (k_0, k_i), is\ a\ 32-byte\ value \\
k_0, k_i\ is\ a\ 16-byte\ value, \ k_i\ is\ a\ 16-byte\ value, \\
n\ is\ a\ 16-byte\ value \\
Salsa20_{k_0, k_i}(n) &= Salsa20\text{Hash}(\sigma_0, k_0, \sigma_1, n, \sigma_2, k_i, \sigma_3)
\end{align*}
\]

The values \(\sigma_0\ thru \sigma_3\) are 4 byte constant values (defined in decimal above). The k value is a user specified key. The n value is a nonce (number used only once). Figure 6-8 shows the Salsa20 expansion function defined in CryptKeeper.
Figure 6-8 The Salsa Expansion Function

The function takes 3 input values, the nonce value \( n \), and the user defined key values \( k_0 \) and \( k_1 \). The \( n \) value is passed to a repeater which is used to trigger the constants \( \sigma_0 \) thru \( \sigma_3 \). The \( N \) node is a repeater node that simply repeats the \( n \) value (used to be explicit). The constants and the input values are then concatenated into one big value \( (\sigma_0,k_0,\sigma_1,n,\sigma_2,k_1,\sigma_3) \). This value is transformed into a 4x4 matrix and passed to the Salsa20 hash function. The output of the hash function is flattened into a scalar value and this value becomes the output of the module.
6.9 Salsa20 Encryption

Now we will explore the culmination of our previous work, the Salsa20 encryption function.

Let \( v \) be an 8-byte nonce

\( Salsa20_k(v) \) is a 2^{70}-byte stream

\[
Salsa20_k(v) = Salsa20_k(v, 0) \oplus Salsa20_k(v, 1) \oplus Salsa20_k(v, 2) \oplus \ldots \oplus Salsa20_k(v, 2^{64} - 1)
\]

where \( k \) is the unique 8-byte sequence \((i_0, i_1, i_2, i_3, i_4, i_5, i_6, i_7)\)

and \( i = i_0 + 2^8 i_1 + 2^{16} i_2 + 2^{24} i_3 + 2^{32} i_4 + 2^{40} i_5 + 2^{48} i_6 + 2^{56} i_7 \)

Plainly stated the Salsa20 encryption function is the concatenation of multiple occurrences of the Salsa20 expansion function. To encrypt using this byte stream we simply xor the byte stream with our plain text message. The byte stream will be truncated to have a length equal to that of the plain text. Figure 6-9 depicts the implementation of the Salsa20 encryption function in CryptKeeper.

![Figure 6-9 Salsa20 Encryption Function](image-url)
For brevity we have only implemented the first 2 iterations of the Salsa20 encryption function. Each occurrence of the Salsa20 expansion function uses the same nonce value and the same key values. Only the constant varies for the two occurrences. The output of each occurrence is concatenated together to produce a 128-byte value. This value is then xor-ed with the plain text (also a 128-byte value). The result of this xor is our cipher text.
7 Conclusion

CryptKeeper was designed with a basic goal in mind, the visual assembly and analysis of cryptographic ciphers. This paper has demonstrated the achievement of this goal. However, in its current state CryptKeeper only handles symmetric ciphers. Functionality can still be added to stretch it abilities even further.

7.1 Future Work

CryptKeeper in its current state is a fully functional application. However as a cryptographic tool, it is still in its infancy. Many exciting and useful applications of CryptKeeper have yet to be discovered. Further development of the system will help it to become a general purpose cryptographic tool.

7.1.1 Beyond Symmetric Ciphers

Extending the power of CryptKeeper to allow it to assemble asymmetric ciphers is an obvious next step in the development of the system. But development could be taken even further. Cryptography has many areas in which visual assembly and analysis would be very beneficial. Visualization of MAC algorithms, and hash algorithms are two examples.

7.1.2 Node Library Extension

Cryptographic Ciphers come in a wide variety of flavors. CryptKeeper has been given basic functionality to handle assembly most of ciphers. However there are times that the node library provided, while sufficient, can be somewhat inconvenient. Adding the ability for users to define their own custom code nodes could be highly beneficial.
7.2 Closing Thoughts

Supplementing the standard approach to learning cryptography with a visual system has great benefits. It gives students the advantage of physically watching the data flow throughout the algorithm. More time can be spent studying the details of these algorithms, rather than spent struggling with details of the algorithms implementation. This approach eases the learning curve for a prospecting cryptographer and has the potential to make the field of cryptography more main stream.
Bibliography


Appendix A

The DES SBox Permutation Tables

| S1     | 14  | 4  | 13 | 1  | 2  | 15 | 11 | 8  | 3  | 10 | 6  | 12 | 5  | 9  | 0  | 7  |
|--------|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0      | 15  | 7  | 4  | 14 | 2  | 13 | 1  | 10 | 6  | 12 | 11 | 9  | 5  | 3  | 8  |
| 4      | 1   | 14 | 8  | 13 | 6  | 2  | 11 | 15 | 12 | 9  | 7  | 3  | 10 | 5  | 0  |
| 15     | 12  | 8  | 2  | 4  | 9  | 1  | 7  | 5  | 11 | 3  | 14 | 10 | 0  | 6  | 13 |

| S2     | 15  | 1  | 8  | 14 | 6  | 11 | 3  | 4  | 9  | 7  | 2  | 13 | 12 | 0  | 5  | 10 |
|--------|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 3      | 13  | 4  | 7  | 15 | 2  | 8  | 14 | 12 | 0  | 1  | 10 | 6  | 9  | 11 | 5  |
| 0      | 14  | 7  | 11 | 10 | 4  | 13 | 1  | 5  | 8  | 12 | 6  | 9  | 3  | 2  | 15 |
| 13     | 8   | 10 | 1  | 3  | 15 | 4  | 2  | 11 | 6  | 7  | 12 | 0  | 5  | 14 | 9  |

| S3     | 10  | 0  | 9  | 14 | 6  | 3  | 15 | 5  | 1  | 13 | 12 | 7  | 11 | 4  | 2  | 8  |
|--------|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 13     | 7   | 0  | 9  | 3  | 4  | 6  | 10 | 2  | 8  | 5  | 14 | 12 | 11 | 15 | 1  |
| 13     | 6   | 4  | 9  | 8  | 15 | 3  | 0  | 11 | 1  | 2  | 12 | 5  | 10 | 14 | 7  |
| 1      | 10  | 13 | 0  | 6  | 9  | 8  | 7  | 4  | 15 | 14 | 3  | 11 | 5  | 2  | 12 |

| S4     | 7    | 13 | 14 | 3  | 0  | 6  | 9  | 10 | 1  | 2  | 8  | 5  | 11 | 12 | 4  | 15 |
|--------|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 13     | 8   | 11 | 5  | 6  | 15 | 0  | 3  | 4  | 7  | 2  | 12 | 1  | 10 | 14 | 9  |
| 10     | 6   | 9  | 0  | 12 | 11 | 7  | 13 | 15 | 1  | 3  | 14 | 5  | 2  | 8  | 4  |
| 3      | 15  | 0  | 6  | 10 | 1  | 13 | 8  | 9  | 4  | 5  | 11 | 12 | 7  | 2  | 14 |

| S5     | 2    | 12 | 4  | 1  | 7  | 10 | 11 | 6  | 8  | 5  | 3  | 15 | 13 | 0  | 14 | 9  |
|--------|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 14     | 11  | 2  | 12 | 4  | 7  | 13 | 1  | 5  | 0  | 15 | 10 | 3  | 9  | 8  | 6  |
| 4      | 2   | 1  | 11 | 10 | 13 | 7  | 8  | 15 | 9  | 12 | 5  | 6  | 3  | 0  | 14 |
| 11     | 8   | 12 | 7  | 1  | 14 | 2  | 13 | 6  | 15 | 0  | 9  | 10 | 4  | 5  | 3  |

<p>| S6     | 12   | 1  | 10 | 15 | 9  | 2  | 6  | 8  | 0  | 13 | 3  | 4  | 14 | 7  | 5  | 11 |
|--------|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 10     | 15  | 4  | 2  | 7  | 12 | 9  | 5  | 6  | 1  | 13 | 14 | 0  | 11 | 3  | 8  |
| 9      | 14  | 15 | 5  | 2  | 8  | 12 | 3  | 7  | 0  | 4  | 10 | 1  | 13 | 11 | 6  |
| 4      | 3   | 2  | 12 | 9  | 5  | 15 | 10 | 11 | 14 | 1  | 7  | 6  | 0  | 8  | 13 |</p>
<table>
<thead>
<tr>
<th></th>
<th>S7</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>11</td>
<td>2</td>
<td>14</td>
<td>15</td>
<td>0</td>
<td>8</td>
<td>13</td>
<td>3</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>11</td>
<td>7</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>10</td>
<td>14</td>
<td>3</td>
<td>5</td>
<td>12</td>
<td>2</td>
<td>15</td>
<td>8</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>11</td>
<td>13</td>
<td>12</td>
<td>3</td>
<td>7</td>
<td>14</td>
<td>10</td>
<td>15</td>
<td>6</td>
<td>8</td>
<td>0</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>13</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td>5</td>
<td>0</td>
<td>15</td>
<td>14</td>
<td>2</td>
<td>3</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>S8</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td>15</td>
<td>11</td>
<td>1</td>
<td>10</td>
<td>9</td>
<td>3</td>
<td>14</td>
<td>5</td>
<td>0</td>
<td>12</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>13</td>
<td>8</td>
<td>10</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>0</td>
<td>14</td>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td>12</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>10</td>
<td>13</td>
<td>15</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>14</td>
<td>7</td>
<td>4</td>
<td>10</td>
<td>8</td>
<td>13</td>
<td>15</td>
<td>12</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>